Open-Source Databases, Part III: Choosing a Database

Reuven M. Lerner

Abstract

Which database is right for you? MySQL or PostgreSQL?

If you are an application developer, you're probably working with large quantities of data. And, if that data is anything more complex than a hash table, you might want to consider moving some or all of it into a relational database. Relational databases are designed for reliable and flexible retrieval of data. The magic of a relational database is not the use of two-dimensional tables to store all of the information, but it's the fact that tables can be combined in many different ways and manipulated using the SQL query language.

As we saw in my database articles in the last two issues of LJ, open-source programmers are fortunate enough to have several database options at their disposal. By far, the two most popular open-source relational databases are MySQL and PostgreSQL. Each has a large and loyal following, and each continues to improve with every successive version.

And, when I write “large and loyal following” above, I'm not kidding. MySQL and PostgreSQL have long been at the center of a major flame war within the Open Source world. If someone on Slashdot dares say something about one of these products, you can be sure it won't be long before someone writes a nasty (and often childish) note about the other one. These disagreements often reflect the knee-jerk attitudes of uninformed users, but there have been no shortage of attacks from well-known and informed users of these products as well.

I believe there are circumstances when either MySQL or PostgreSQL might be an appropriate choice. I've strongly preferred PostgreSQL in my work during the last decade—yet, there definitely are times when MySQL seems to be the more appropriate solution.

So, despite my personal biases and the risk of opening a flame war within the Open Source community, I now conclude this series about open-source databases with a comparison between MySQL and PostgreSQL in a number of different categories. I hope by the time you finish reading this article, you understand that choosing a database is almost never a matter of finding the “fastest” or “best” product, because there is no one way to measure the quality or appropriateness of a relational database server. Rather, I hope you'll be able to consider each of these on the basis of its own merits, rather than on the propaganda that is so widespread.

Data Integrity

Perhaps the first and foremost task of a database is to store and retrieve data reliably. Just as you wouldn't want to use a hard disk that occasionally loses data, you don't want to put things into a database...
that occasionally mangles its contents. This is true even if the reliability comes at the expense of speed.

The gold standard for reliability in the database world has an acronym, ACID (Atomicity, Consistency, Isolation and Durability). This means that under all circumstances in the database, the following hold true:

- **Atomicity**: each query is guaranteed to complete or not, without any possibility of halfway or incomplete states.
- **Consistency**: the database is always in a legal state before and after a transaction.
- **Isolation**: each transaction occurs separately from other actions, so that you can't have two transactions interfering with one another.
- **Durability**: transactions persist over time, typically by being stored on a filesystem.

The attitude toward ACID within the PostgreSQL community has been unchanged since I first started to use it a decade ago, placing it as the highest possible priority. This doesn't mean PostgreSQL is lacking in other features, but rather it means the developers have worked to ensure that data stored in a PostgreSQL system will be consistent and reliable, even if you do nasty things such as issue a `kill -9` or pull the plug.

During the past few years, PostgreSQL has begun to offer even better support for transactions and database stability, using write-ahead logs (WALs) that describe each action taken by the database. These WAL files can be used to recover from a disaster or even to recover the database to an earlier point in its history—a feature known as point-in-time recovery (PITR). Thus, if you know something happened yesterday, but the database was working perfectly two days ago, you could use PITR to recover to the earlier, stable state. Recent versions of PostgreSQL also support two-phased commit, a type of transaction you're likely to see in a distributed system where multiple servers must coordinate their actions.

MySQL has had a mixed attitude toward ACID during the years. When I first started to use MySQL in 1995, the authors' attitude was that transactions should be handled by the application, not the database. Indeed, as recently as 2000, the to-do list for MySQL included tasks having to do with production-quality transaction-safe tables. This has led to a great deal of bad blood between the MySQL and PostgreSQL communities, with members of the latter sometimes claiming that no critical data should ever be stored in MySQL.

The good news is that modern versions of MySQL do indeed support transaction-safe tables, using InnoDB, a third-party product released under the GPL that has been integrated into MySQL for several years. Moreover, InnoDB appears to use techniques that PostgreSQL and Oracle have used for years, such as MVCC (multi-version concurrency control). The bad news is that at least some benchmarks I've seen indicate that InnoDB has some problems scaling to large numbers of simultaneous queries.

In addition, the company that develops InnoDB recently was bought by Oracle, which might lead some people to worry about future licensing, development and pricing issues. For the time being, this latter issue does not appear to be a serious one, because Oracle and MySQL signed a contract in 2006 extending the licensing for InnoDB. But, MySQL does not appear to be taking any chances and has hired several experts to create a new table structure that will be owned by MySQL and thus be impervious to
such business problems.

I'm personally of the persuasion that true ACID compliance is always a good thing to have around, much like seat belts in a car. Sure, you can drive without a seat belt, and the odds are that nothing will happen to you. But, it's impossible to predict when something bad might happen, and you really don't want to be without a seat belt under such circumstances. In the same way, if your data is important to you, it's best to ensure that it will persist with integrity.

A related problem has to do with the degree to which each database enforces constraints and limits. PostgreSQL tends to be quite stringent on such matters, refusing to accept illegal data. MySQL tries to be more forgiving and flexible, but that can result in strange and illegal data being stored.

For example, consider the following set of MySQL commands, in which we create a table foo with a single column (named a) of type DATE: Garrick, small font below.

```
mysql> CREATE TABLE foo (a date);
Query OK, 0 rows affected (0.08 sec)

mysql> INSERT INTO foo (a) VALUES ('2007-feb-30');
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> SELECT * FROM foo;
+------------+
| a          |
| 0000-00-00 |
+------------+
1 row in set (0.00 sec)
```

By contrast, this is what happens in PostgreSQL:

```
atf=# CREATE TABLE foo (a date);
CREATE TABLE

dat=# \d foo
Table "public.foo"
  Column  | Type  | Modifiers
          |-------|-----------
    a      | date  |
/dat=# INSERT INTO foo (a) VALUES ('2007-feb-30');
ERROR:  date/time field value out of range: "2007-feb-30"
```

It is possible to configure MySQL to be more strict on such issues, but most users will not think to do so and will be stuck with illegal values in their tables.

Given the political and technical issues at MySQL, as well as the weird (and potentially dangerous) default behavior in MySQL, I believe that PostgreSQL has a big edge on issues of data integrity.

**Features**

MySQL and PostgreSQL offer a very large number of built-in features, many of which have been added in response to community requests and reactions. Both offer a large number of data types, which can be mixed and matched within a single row without restrictions. (The very limiting row-length restriction
that plagued versions of PostgreSQL is now ancient history, I'm happy to say.) Both databases now support Unicode characters; MySQL supports both UCS-2 and UTF-8 encoding, and PostgreSQL supports only the latter.

Both databases also offer a very large number of functions that manipulate data, including strings and dates. It's quite convenient to be able to compare and sort dates or to find all rows whose timestamp was within the last 24 hours. PostgreSQL's interval data type, which describes a length of time (rather than a particular point in time), has proven to be particularly useful. MySQL has a number of different types that database purists like myself frown upon, such as SET and ENUM, but that are undoubtedly popular for many users.

In many areas where PostgreSQL has had an advantage, MySQL is beginning to catch up. PostgreSQL users have long been able to create new data types and functions that operate on those types. Indeed, PostgreSQL offers developers the unusual ability to write server-side functions in a number of languages, including SQL, Perl, Python, Java, Tcl and the R statistical language. MySQL does not allow for the creation of new data types, but recent versions do provide the ability to write server-side functions and stored procedures.

MySQL has offered a built-in solution for full-text search, accomplished by using a special type of index on text fields. However, there are some important restrictions on this index, such as the fact that it works only with MyISAM tables. Given that these tables support neither foreign keys nor transactions, I am a bit nervous about suggesting them as a solution.

PostgreSQL's full-text search solution (tsearch2) has the opposite problem. Although it is robust and works well within PostgreSQL's standard transactional tables, it requires some work to configure and install. Most administrators and programmers will be able to install it successfully within a short period of time, but nonetheless, there is a difference between a built-in capability and one that needs to be added.

PostgreSQL has a number of built-in features that MySQL either has yet to implement or that are scheduled for future releases. Among these are the ability to use subselects anywhere in a query, the use of sequences (rather than simple auto-increment columns), rules that allow users to modify the way queries are interpreted on a given table and CHECK constraints on column values. Recent versions of MySQL now include features that were previously available only in PostgreSQL, such as triggers and views.

In general, the PostgreSQL development group seems to emphasize SQL standards more than MySQL does, although the MySQL developers appear to be increasingly sensitive to this need and now offer an --ansi command-line switch for those people who want to work in a standards-compatible mode all of the time.

Both MySQL and PostgreSQL are extremely easy to use. Each comes with a command-line client program that is packed with features, allowing you to manipulate your database by sending SQL queries. I have become spoiled by some of the features of the PostgreSQL command line, such as the expanded output (\x).

The command-line interfaces for both databases have grown more useful over time. Although the MySQL interface might appear to have fewer commands, that's partly because MySQL has made some data available via SQL queries (for example, SHOW TABLES), which would require more complicated
queries in PostgreSQL, leading to the creation of a shorthand command, \dt. Both command-line interfaces use GNU readline, making it easy to edit and re-issue queries. Both also allow users to edit the previous query using the `e` command.

Overall, it's probably fair to say that PostgreSQL offers a superset of MySQL's capabilities, aside from a few issues (for example, built-in text indexing). Those capabilities that PostgreSQL does not have, such as new data types and functions, are added into the system easily, without needing to recompile or otherwise modify the core PostgreSQL server. That said, I believe MySQL's capabilities are nothing to sneeze at and are likely more than adequate for most applications you might be writing.

**Administration**

Both MySQL and PostgreSQL are amazingly easy to administer, especially in small- and medium-size cases. You (optionally) change a few configuration options, start the server and then walk away. There's really not much more to do than that. For anyone who has worked with a larger database system, such as Oracle, this is a refreshing change. However, there are slight differences in the ways the two systems operate.

PostgreSQL relies on several external UNIX-level commands to create and manage databases and users, as well as the activity of the PostgreSQL server. There is no central PostgreSQL administrative program. MySQL, by contrast, has a central `mysqladmin` program that handles most functions having to do with server startup and shutdown, as well as the creation and destruction of databases. The creation and management of users is handled by manipulating tables in the `mysql` database.

PostgreSQL's counterparts to the `mysql` database are special system tables and views, all of which begin with the `pg_` prefix. These tables, although necessary for the system to run, easily can be ignored by most programmers and come into play only when trying to tune the system or figure out how to optimize queries.

GUI-based administration tools are available for both programs, as well as Web-based tools written in PHP. To be honest, I haven't used these tools much during the years, given my familiarity with (and preference for) command-line systems for working with databases. However, my experience with both sets of GUI programs has been positive, and my impression is that they are both stable and secure, as well as useful.

Another aspect of administration unique to PostgreSQL is the need to “vacuum” dead rows from the database to return them to the operating system or to other rows that could benefit from the space. In addition, PostgreSQL's vacuum function visits the rows of the dead and uses the statistics it collects to inform the optimizer and query planner. Nowadays, the auto-vacuum daemon takes care of this automatically for most people, removing the long-dreaded need to schedule it in cron.

*One administrative area that is particularly hot right now is replication.* Many Web sites and other applications are pushing the limits of their database servers, and it would be useful to split the work among multiple servers. Of course, this raises issues of data integrity and synchronization among distributed processes. The simple solution to the problem is to have a master/slave relationship among the different servers, with UPDATEs and INSERTs taking place only on the master server, and SELECTs taking place on the slave servers. Solutions for this exist under both MySQL and PostgreSQL, although the PostgreSQL solution (Slony) is external to the standard package and apparently can be
difficult to install and configure.

A more complicated setup involves the use of two master database servers. MySQL appears to have taken the lead on this front with a relatively new clustering tool. But, PostgreSQL users, who have been clamoring for such tools for several years now, appear to be on the verge of getting their wishes fulfilled.

Finally, no database server would be worthwhile if it weren't possible to perform regular backups. pg_dump and mysqldump are command-line programs that turn the current contents of a database into a text file. Such dump files are quite useful and can be used to rebuild the database when necessary.

I would argue that when it comes to administration, the two database products are identical—unless you need replication, in which case you'll probably benefit from MySQL's greater experience and replication integration.

**Performance**

For years, one of the claims made in the MySQL/PostgreSQL flame war has had to do with speed. MySQL fans often have claimed that their system is faster, particularly for read-only tasks, making it a superior choice for Web sites where most data is read. PostgreSQL advocates, in contrast, claim that their system holds up to big loads much better than MySQL.

I haven't conducted any benchmarks of my own, but my reluctance to do so is an admission that I'm unqualified to create a good benchmark, and not that I believe the two systems are identical or that performance isn't important. Moreover, as I stated previously, I believe that performance is secondary to data integrity. I would much rather have a slow, reliable database than a fast one that occasionally will wreak havoc on my data.

From the benchmarks I've seen, it appears that MySQL is indeed faster than PostgreSQL when working with a small number of clients or with read-only data. However, all of the comparisons I've seen over the last few years indicate that as more clients are added to the system, PostgreSQL handles the load better.

Does this mean that PostgreSQL always will be faster? Of course not. But, it does mean that on particularly popular sites, PostgreSQL may hold up better.

Maybe I'm simply naive, but I decided several years ago that I would largely ignore the performance debate when it came to databases. Both MySQL and PostgreSQL have large followings and have been used on large-scale systems. The data seems to indicate that PostgreSQL has an advantage, but enough people are using MySQL on large Web sites that I have to assume it is working well enough for them.

**Support**

Finally, no comparison would be complete without mentioning support. We might consider several types of support—from the strength of the Open Source community to the number and quality of companies supporting (and developing) the software to the number of third-party applications that support each database.

It is impossible to ignore the extremely large number of MySQL users in the world. This has led to an outpouring of books, tutorials and mailing lists for MySQL—some (but not all) of which have been
sponsored by the MySQL company itself. If the community-based support is not enough, it is possible to buy commercial support for MySQL from a number of companies, including MySQL AB.

PostgreSQL has a smaller community, and a smaller number of books and tutorials available. However, my experience has been that the community is responsive to questions and suggestions, and that the lead developers often are quite willing to answer questions from all levels of users.

Many open-source packages support both MySQL and PostgreSQL. But, it is rare to find a package that supports PostgreSQL exclusively, and it is easy to find packages that support MySQL alone. This has been a source of some frustration for members of the PostgreSQL community; however, there doesn't seem to be much anyone can do about it, short of asking for patches or contributing such patches.

A recent thread on the main PostgreSQL mailing list asked about CRM packages that support the database. Although there were a few, there was definitely some grumbling about the lack of PostgreSQL from other open-source projects. Those projects often are staffed by small groups of volunteers who rarely understand how they can make their SQL more portable and thus easier to use on multiple brands of databases.

The bottom line on support is that although PostgreSQL support is excellent, MySQL support is overwhelming. If there is a winner here, it's MySQL.

Conclusions

So, should you pick MySQL or PostgreSQL for your next database task? All things being equal, I strongly recommend PostgreSQL. Its community might be smaller, and there are fewer resources available in print and on the Web. But, it has more features to ensure data integrity, its features are largely a superset of MySQL, and it always offers transactions and referential integrity, without having to specify a particular type of table.

That said, there are reasons to use MySQL: if you already are using it, if you need commercial or community support, if you need replication, or if you are using software that is incompatible with PostgreSQL, MySQL is a fine choice. Just make sure to use InnoDB tables, so that you can take advantage of what a database always was meant to do—ensure the quality of the data.

Resources

The PostgreSQL home page is [http://www.postgresql.org](http://www.postgresql.org). Similarly, the MySQL home page is [http://www.mysql.org](http://www.mysql.org). Each has its latest manuals posted, as well as software, drivers and discussion lists.

A table comparing administration and programming of the two databases is available at [http://linuxboxadmin.com/articles/postgresql-for-mysql-users.php](http://linuxboxadmin.com/articles/postgresql-for-mysql-users.php).

A relatively recent comparison of the two databases' performance is at [http://www.mysqlperformanceblog.com/2006/11/30/interesting-mysql-and-postgresql-benchmarks](http://www.mysqlperformanceblog.com/2006/11/30/interesting-mysql-and-postgresql-benchmarks), which points to the following: [http://tweakers.net/reviews/657](http://tweakers.net/reviews/657).

Finally, a comparison between the databases (but perhaps a bit out of date), along with Oracle, was conducted at CERN, the European center for particle physics, and is available at